Query-Based Change Detection

Tony C. Carnes and Eric Hanson
301 CSE
CISE Department
University of Florida
Gainesville, FL 32611-6120
(352)-392-2730
tcc@cise.ufl.edu
http://www.cise.ufl.edu/~tcc

TR-98-015; Version 1
22 October, 1998

Abstract

This paper presents several algorithms that can be used to efficiently isolate changes in queriable data sources. The introduction of external trigger systems and the popularization of data warehouses have served as a catalyst for identifying efficient change detection techniques. Here we propose multiple algorithms for change detection as the optimal approach is determined by the capabilities of the underlying data source. We focus on simple and legacy data sources that support querying but do not necessarily support other methods that allow for more efficient change detection, such as triggers or asynchronous replication. We argue that most reasonable sources of data will have some structure and are at least queriable through some form of SQL-enabled gateway. Utilizing this fact, our work improves upon previously developed differencing algorithms by "pushing down" restrictions (reducing I/O), by comparing only interesting attributes (reducing CPU), and by incrementally comparing subsets where necessary (reducing locking). Combining these techniques enables frequent differencing on data sets of all sizes.

1. Introduction and Related Work

Change detection in data repositories has recently become a topic of renewed interest due to the popularity of data warehouses and the need for replication in heterogeneous environments. Identifying and applying changes to snapshots (delta-replication) instead of replicating entire relations can greatly reduce the time required to refresh a snapshot. While internal triggers are traditionally recognized as one of the most cost-effective means of isolating changes, their availability is limited to only the most recent database systems. Trapping an update before it is applied to the database also yields reasonable performance, but introducing an intermediate layer or modifying the application code is not always a viable option [Usoft96]. Efficient change detection techniques are needed for those systems that are not enhanced with either of these capabilities.

An alternative approach, proposed by Chawathe, [Chaw96, Chaw97a, Chaw97b] suggests comparing two versions of the data set to identify change. This technique, known as *differencing*, can be used with most data sources, including both structured and semi-structured data. Although differencing is not traditionally recognized as an optimal solution for isolating changes in data sets, it does provide a reasonable option for environments where more efficient mechanisms, such as triggers and replication, are not available [Lind86, Labi96]. The main pitfall of differencing is the relatively high cost of each relation comparison. Previous attempts at reducing differencing costs on structured data use sorting and compression techniques at the local site to lessen the I/O required to compare multiple copies [Chaw97b][Labi96]. Minimizing the costs of gathering changes in semi-structured data is

accomplished by using lowest-cost edge detection algorithms where an edge is the step necessary to transform one aspect of the old image to the new image [Chaw97a][Chaw96]. The techniques used in both instances do not yield sufficient performance for commercial applications due to I/O and locking constraints, as will be demonstrated in section 5.

The basic tenant of differencing is to use some type of difference command, like the Unix or VMS "diff," to determine the difference between an older copy and the current data set. Given a data set of size *n* pages, the space complexity for a simple differencing system is at least *2n, n* pages for both the original and the replicate. As the algorithm must compare corresponding rows between the two data sets, the I/O complexity required to identify all of the data changes can be as high as n^2. If insufficient memory exists to contain $n+1$ pages, each page reference of n can require an entire table scan of the replica, thus yielding n^2 I/Os.

Previous differencing algorithms seem to have ignored the fact that change detection is a two-stage process. First, the actuality that something in the set (tuple, data set, or image) has changed must be recognized. Second, the nature of the change must then be isolated. The existence of a change implies that the data source is actively being modified. The frequency at which the differencing algorithm is run greatly affects the expected number of changes detected at each run. Earlier estimates of a 20% change in data between comparisons [Labi96] are valid for relatively few business applications, even if the algorithm is run once a day. A more realistic estimate is a 1-% change per day for a given relation.

If differencing is deferred until the system is quiet, perhaps the previous algorithms proposed by Chawathe and Labio are sufficient. Not all change detection can be postponed until the system is inactive, however. If modifications must be detected shortly after they occur (i.e. for use with an asynchronous trigger processor [Hans97]), the basic differencing algorithm is not adequate, especially for relatively large data sets. In addition, writing differencing applications for each source of data is not sufficiently cost effective to make it a viable solution.

In an effort to improve concurrency, reduce I/O, lessen space complexity, and eliminate the need to write a differencing application for each data source, several new algorithms are introduced here. These algorithms are designed to work with relational and object-relational database data sources. As a practical matter, they can be implemented for any ODBC-compliant [ODBC94] data source of which there are more than 125 currently available. The introduction of *several* query-based techniques instead of only one enables full utilization of the capabilities of each data source.

ODBC compliant data sources generally fall into three broad categories – those having modifiable table schema definitions, those with modifiable database schema definitions that disallow changes to existing table schemas (i.e. new tables can be created), and those that are queriable where neither the table nor the database schema is modifiable. If the source data repository can perform joins *and* allows a duplicate copy of the data to be maintained at the same site, the source data server can be used to detect appropriate changes. If either of these conditions fail, then the original, or some part of the original, must be forwarded to a separate remote site to maintain the copy. It is assumed that the remote site can perform joins. In both the single and remote site scenarios, the following algorithms represent several query-based approaches that can be easily implemented to create realistic differencing applications against each category of data source. The basic algorithms are outlined below. These algorithms are further explained in later sections.

- **Modified Table with Time-stamps (TS)** – For use with systems where the underlying "relation" can be augmented with a time-stamp attribute. This

algorithm yields good performance for insert and update-only systems. It is especially applicable for systems that can automatically maintain the time-stamp attribute without requiring application modification.

- **Modified Table with Time-Stamp and Delete Fields with Non-clustered (TDI) or Augmented Non-clustered Index (ATDI)** – For use with systems where the application code and the base relation can be modified. Both TDI and ATDI introduce a secondary index to eliminate the need to scan the base relation to identify changed tuples, thus reducing the number of records that must be locked. Augmenting the secondary index with the appropriate attributes (ATDI) eliminates the need to join back to the base relation and can be proven to yield optimal I/O cost if the data source supports compression. This algorithm quickly identifies changes with minimal I/O and does not lengthen on-line transactions. The application code must be modified to make use of the new time-stamp and delete attributes.
- **Modified Table with Tuple Level Checksum (TLCS)** – For use with systems where either the table schema or the database schema can be modified but the application code cannot be changed. Instead of the use of a time-stamp attribute to help identify which tuples have been modified, a checksum is calculated and maintained with each tuple. A separate process scans each relation, comparing the tuple to the checksum. If a discrepancy is identified, the tuple is flagged as having been modified. This approach does not perform well against large systems that allow *deletes* as a full sort merge join with outer join detection must be used to identify deleted rows, thus negating the benefit of the checksum.
- **Modified Database Schema (MDS)** – For use with data sources where the database schema can be modified but the relation schema cannot be changed. To identify changes, a join and outer-join of the two relations (old and current copy) must be performed. Excellent performance can be realized if the relevant snapshot, the subset of attributes and tuples, is highly restrictive. The worst-case scenario of this technique yields the same performance characteristics of the best-case previous algorithms. This technique can induce locking problems for large, non-restrictive snapshots, as the two relations must be scanned.
- **Local Site Incremental Differencing (ID)** - For use with extremely large data sets where the database schema can be modified and sufficient space is available on the local system. Similar in technique to MDS, except that ID eliminates locking contention through incrementally differencing subsets instead of comparing the entire relation. Although I/O is comparable to the best-case older algorithms, this technique does not suffer from a serious lock contention problem.
- **Remote Site Incremental Differencing with Range Checksum (ID/RCS)** – For use with systems where the local site cannot be modified or insufficient space is available at the local node. This algorithm combines a variant of the techniques used in TLCS, MDS, and ID. An incremental join is performed against the old and new copy, where the new copy is relocated to a remote site. To reduce the number of tuples of the relation at the source site that must be copied to the remote site before comparison, a checksum is assigned to each range of data of the source site instead of to each tuple. The range checksum is maintained at the remote site and is used with the query of the source site to determine if a change has occurred somewhere within a given range. The incremental nature of this

algorithm eliminates the locking constraints of other algorithms. An example of this technique is found in section 3.6.

- **View-based Differencing (VBD)** – For use with systems where snapshots are based on a particular view, as in data warehouses. Variants of this technique can be used to greatly improve refresh costs of the data warehouse view by isolating changes at the view level instead of the relation level.

As each algorithm is most appropriately used with a given class or category of data source, the algorithms are separated by the distinguishing characteristic of the data source – query only, modifiable database schema, and modifiable table schema – in later sections. To ensure a better understanding of the usefulness of each algorithm, a brief description of the key issues related to differencing-based change detection precedes the algorithm evaluations.

2. Differencing Issues in OLTP Systems

Determining changes in OLTP systems necessitates addressing concurrency issues. In almost all structured data systems, a read-lock is imposed on a tuple or page when data is read into memory. When scanning tuples in a system where data is stored as relations, locks must typically escalate from page or row level to a table level lock if roughly more than ten pages are accessed [Ingr91]. (This is often tunable, but here we are talking about the general case.) The loss of concurrency caused by the acquisition of a large number of locks during a differencing scan can prohibit a differencing technique from being utilized on a frequent basis if locking is not properly addressed. Frequent change detection is critical for both Asynchronous Trigger Processors (ATP) and systems where the amount of difference between two replicate snapshots is restricted.

Many algorithms, like sliding-window and pre-sorted sort-merge [Chaw97a] [Chaw97b][Ull89][Labi96], have been introduced for sorting then comparing a current copy to an older one. In each, the cost of every differencing scan against a data source of size n pages is n^2 for semi-structured data and $2n\log n + n + m$ for structured data. The cost to sort the base relation is 2nlogn and m is the size in pages of the duplicate copy. As the size of the duplicate and the base relation are essentially the same for these algorithms, the cost can be approximated to $2n+2n\log n$ if the copy is already stored in sorted order. This cost model is correct assuming memory is not sufficient to contain $2n$ pages and that the algorithm always uses a sort merge join with outer join detection. With enough memory ($|\text{Memory}| > \text{Sqrt}(n)$), the costs can be reduced to approximately $6n$ [Labi96]. The cost of the comparison can be reduced to $2n$ if a sliding window comparison or a pre-sorted sort-merge technique is used [Labi96]. The sliding window protocol requires the strict assumption that the data cannot be restructured (e.g. modified from a B-tree structure to a hash structure). Pre-sorting techniques require that the relations are stored in sorted order. Requiring static or pre-sorted structure invalidates this option for many data sources.

To force sorted order, a query can be run to create a separate file that is used in the comparison step. Requiring the creation of yet another copy increases the comparison costs of [Chaw97b][Labi96] to $8n$ and increases the space complexity to $3n$. If the underlying relation is not sorted, the comparison cost for large tables is prohibitive. For example, sorting a 200,000-tuple table can take several minutes and will acquire a read-lock on the entire data set. Locking the entire data set during report creation is generally not acceptable for OLTP systems. Removing read-locks from the scan of the base relation improves concurrency but it allows for the introduction of dirty reads that can corrupt the copy of the base relation.

Current update inference algorithms [Chaw97b][Labi96] ignore or do not sufficiently address issues such as locking and excessive I/O. In addition, their applicability for general use is questionable given their inability to determine differences between two sources that are not simply flat files. The existence of escape sequences and format characters in many data sources prevent the use of a simple "diff" command. Even in situations where locking is not critical and the files being compared are flat files, the cost, *2n*, of the best case algorithm proposed by Labio [Labi96] can still be prohibitively high if the difference operation must be run more frequently than once a day. In this paper we outline several new algorithms that are applicable for most structured sources of data (i.e. those that are queriable through a gateway such as ODBC.) The performance characteristics of each algorithm are given along with descriptions of expected locking constraints. These new techniques make use of meta-data and recent technology advancements in heterogeneous connectivity to achieve excellent performance.

3. Query-Based Differencing

The concept of query-based differencing is much the same as the techniques outlined by Labio, Chawathe, Garcia-Molina, and Widom in that two copies are compared to identify some type of change to a data set. Query-based differencing is different, however, in three main areas. First, the comparison is performed through the use of a query language, such as SQL, so the benefit of data-independence, projection, and selection can be utilized in the comparison step. Second, the copy is not required to be an exact duplicate of the original as only pertinent subsets of data are of interest. Finally, pure comparison to identify changes can be augmented with flags and time-stamps, much like [Lind86], to reduce the cost of identifying which tuples have been modified.

Query-based differencing can be accomplished by using SQL with a "gateway" such as ODBC to gather differences from heterogeneous data sources [Geig95]. The large number of ODBC drivers currently available enable relatively seamless access to both structured and semi-structured data sources. The drivers understand a specific syntax of SQL, translate the query to the native SQL of the underlying data source, if applicable, and then either directly or through the native data server gather results from the data source [ODBC94]. The gateway may need to reformat the return values (e.g. date fields) before passing the results back to the front-end application, but the SQL application does not require modification even if the data source is converted to a different DBMS product. Often ODBC drivers for simple data sources are enhanced with database server capabilities to provide both join and optimization functionality.

The query-based differencing approach provides several benefits over simple flat file differencing. As an example, most data sources that contain information of interest are not simply flat files. A UNIX "diff" command cannot be run against two EXCEL or LOTUS spreadsheets because the underlying files contain many special characters that make the comparison difficult. Exporting data to a flat file before analysis can simplify testing but it is inefficient and completely unnecessary given recent advances in ODBC technology. As such systems are ODBC compliant, the appropriate driver can be utilized to perform an SQL join operation of the original data set with an older copy (or one of several other techniques listed in later sections) to determine the differences. An SQL-based technique does not require the pre-sorted or static structures indicated in previous algorithms but instead uses the optimizer of the data source to determine the best search strategy. The optimizer often has knowledge

of selectivity factors and indexes that can be used to help reduce the cost of change detection. The use of a local server can further reduce the cost of subsequent comparisons as previously cached data may remain in the server. In addition, the appropriate use of SQL allows for data independence so that schema modifications do not "break" the differencing program.

Not all ODBC compliant data sources provide the same functionality, however. A data source may allow changes to relation schemas, the database schema, or it may disallow any type of change. In order to take full advantage of the capabilities of each data source, the SQL-based approach provides a suite of algorithms that can be selectively used in different environments. Table 1 outlines the variables that are used to evaluate each algorithm.

Notation	Definition
ATP	Asynchronous Trigger Processor
TS	Time-stamp attribute
DF	Delete Flag attribute – one bit or byte depending upon implementation
B	Base relation used for comparisons
B_a	New B augmented with TS&DF
C	Copy of B excluding changes occurring since the last difference command
C_r	Copy of B restricted to pertinent attributes and tuples
X	Number of Tuples in *B*
Y	Number of Tuples in C_r
TDI	Time-stamp with delete flag algorithm with secondary index
ATDI	Time-stamp with delete flag algorithm with augmented secondary index
TLCS	Modified table with Tuple Level Checksum algorithm
MDS	Modified Database Schema algorithm
ID	Local site incremental differencing algorithm
ID/RCS	Incremental differencing with range-value checksum algorithm
VBD	View-based differencing

Table 1: Definition of Terms

Variable	Definition	Default Value
S	Size of B_a in Pages. $S = X/(P/(T+A))$. It is possible to increase the tuple size without increasing table size if *mod(P,T)>(P/T)*A*	1.1 N
M	Size of C_r in pages = Y/(P/(size(key)+F))	.3 N
K	Size of Key in Bytes	8 bytes
F	Sum of the size of applicable fields requiring change detection	
N	Size of *B* in Pages	1000 pages
P	Page Size in Bytes	2048 bytes
T	Tuple Size in Bytes	200 bytes
A	Size of TS+DF in bytes	13 bytes
I	Size of Secondary Index on *B* in Pages, ie. $X/(P/(A+size(tid)))$.	.1N
R	% of rows applicable for change detection	100%
Δ	Percentage of table to change between scans	1%
Z	Number of relations involved in VBD join	4
K	Constant – height of index	3

Table 2: Definition of Variables with Default Values

3.1 Modified Database-Schema Approach (MDS)

If the local site can be modified to contain a pseudo-duplicate copy of the base relation, no change to the original or the application program is needed. If the local site data

server has a moderately sized cache, and the difference operation is performed against small tables at fairly frequent intervals, it is possible for both the original and the duplicate relation to remain cached in local server memory. In this case, the cost of performing subsequent joins is 0 IO and minimal CPU usage, which is better than the *2N* IO required for each invocation in the best case snapshot scheme as the previous techniques must read data from a file before comparing. Figure 1 shows the basic algorithm for using SQL to determine the differences between the original – *basetable* with attributes *key, t1, t2,…,tn* – and the copy – *duplicate* with attributes *key, t1, t2,…,tm.* Note that the duplicate may be a subset of the base relation if only certain attributes have relevance to the desired snapshot.

```
Select change_type='Update',t1.key,t1.f1,t1.f2,…,t1.f_m from base-table t1,duplicate t2
Where t1.key = t2.key
   And (t1.f_a != t2.f_w or t1.f_b != t2.f_x or t1.f_c != t2.f_y or … or t1.f_n != t2.f_m)
 {And Restriction(t1)}
Union
Select change_type='Insert', t1.key,t1.f1,t1.f2,…,t1.f_m from base-table t1
Where not exists (select t2.key from duplicate t2 where t2.key = t1.key)
  {And Restriction(t1)}
Union
Select change_type='Delete', t2.key,t2.f1,t2.f2,…,t2.f_m from duplicate t2
Where not exists (select t1.key from base-table t1 where t1.key = t2.key)
  {And Restriction(t1)}
```

Figure 1: Simple Query-Based Differencing Algorithm (QDA)

SQL-92 simplifies this query with the full outer-join operator, but the algorithm as stated is designed to be as generic as possible and as such uses UNION instead of the SQL-92 OUTER-JOIN operator. The SQL algorithm restricts comparisons to the pertinent attributes and tuples of the snapshot by modifying the fields referenced in the SELECT and WHERE clauses. In addition, if relevant changes are restricted to a subset of the tuples, the WHERE clause can be modified to select only appropriate rows from the original relation. As an example, *Restriction (t1)* above could be *t1.dept='ABC'*. The algorithm can be easily generalized to work with any SQL-based system. If the number of updates is high in comparison to the ones that apply to the snapshot, these capabilities can greatly improve performance by reducing the number of spurious messages forwarded by the detection algorithm. The reduction in the number of unnecessary messages can be quite significant.

As an example, an externally defined trigger or a replicated copy (snapshot) may only be interested in *salary* changes (size *8* bytes) to persons in department *ABC*. The *Personnel* table, *P*, is comprised of *21* attributes with a total tuple size of *200* bytes and a key size of *8* bytes. On average, approximately *10* records fit per page. Assuming *10,000* employees, the size of P, designated *N*, will be *≈1000* pages (ignoring index overhead). Also, given the assumption of equal distribution between *100* departments, there will also be *≈100* persons per department. Finally, assume field changes are equally distributed between all non-primary-key attributes.

With the pre-sorted and sliding-window algorithms, the cost of comparison, even assuming presorted data, is at least 2*N* and is more likely *8N* as *P* cannot be required to be sorted for most systems and flat files are used during comparison. In addition, of the number of detected changes, very few are actually applicable (only those pertaining to dept ABC). In this example, the number of applicable changes detected is

.01 (only department ABC).05(only salary attribute)=.0005 of reported changes are of interest*

Assuming each detected change is forwarded through a message to the requesting agent, [Chaw97b][Labi97][Lind86] all send messages where 99.95% are of no value. Finally, although CPU is much faster than I/O, a CPU bound system will be hindered by comparing entire tuples instead of selected attributes. As with messages, approximately 99.95% of the CPU used for comparing attributes is performing useless work. With the SQL-based algorithm, benefits in space, time, and message complexity can be realized.

A further improvement in space and time complexity can be realized by restricting the "copy" to pertinent records and attributes. Given the estimations above, the "copy" of the base relation need only contain the key and the salary fields for a total tuple size of *16* bytes. In addition, the total number of applicable tuples from *P* is *100*. The total size of the copy is then *1600* bytes, or *1* page. Even without a non-clustered index on *P*, the total I/O cost for each differencing run is at most *N+1* due to the relatively small size of the restricted copy. As the copy also contains the primary key, the total I/O cost for joining back to *P* to detect deletes and updates should be at most *.01(N)+1* if the base relation is indexed on the primary key. If a secondary index is available on the attributes of P that are used to restrict the number of tuples in the snapshot, the cost for insert detection can also be reduced to *.02(N)+1*. Thus I/O is at most *N+1* and may be as low as *.03(N)+2*. Only changes that are of interest are reported, thus, in this example, the total number of messages is *99.95%* fewer than previous techniques.

Query-based differencing can further improve performance by allowing the underlying database system to use its full query processing capabilities. As an example, B-trees with a height of 3 and 3 data pages typically require 5 I/O to read data into memory and thus will cost 10 I/O per relation comparison. For base tables of 3 data pages or less, leaving the underlying tables as heap instead of B-tree can reduce I/O to 6, thus reducing I/O by 40%. Although CPU utilization will be higher in the second case, this tradeoff will typically lead to a 35% improvement in overall system response time. Allowing the DBA to choose storage structures and allowing the optimizer to choose the best join strategies can improve upon the forced implementations of storage structures required by pre-sorted sort-merge algorithms.

The worst case cost of the MDS algorithm occurs when the data source does not support some type of sorted structure, the replica contains all tuples and attributes, and the relations are to large to fit into memory. In this case, the optimizer will most likely choose to sort the base relation before joining to the older copy. Although the costs to sort then compare are less than generating a report due to the fact that no output file is created, the cost to perform this operation can be as high as 4N. While some portion of the older copy and the base relation may reside in memory, thus slightly reducing costs, the worst case cost is 4N. In such an environment, however, the best case cost of the sliding window protocol is also 4N. If key-valued secondary indexes are available and the memory is of sufficient size to hold *N+1* pages, the total I/O cost can be reduced to 2.2 N if |each index| is .1N (used to determine *inserts* and *deletes*) and a hash join is used to compare relations for *updates*.

For small tables and highly restrictive snapshots the basic SQL-based algorithm can be used to improve the performance of previous algorithms. But what about tables spanning hundreds of pages or data sources without join capabilities? Data sources without join capabilities will be addressed in section 2.3. For large relations with few restrictions, the bottleneck of locking the base relation while performing differencing must be addressed. Most of the previous algorithms ignore this issue and thus lead to errant performance characteristics for OLTP systems. The SQL-based approach offers several solutions by either eliminating the need to scan the entire base relation or by incrementally referencing subsets of

the data. The level of improvement realized over previous algorithms depends upon the chosen algorithm.

3.2 Modified Table-schema Approach

Reducing the cost of scanning the original relation can be accomplished in several ways. If the number of pertinent tuples is small and if a secondary index is available on the attributes that are used to restrict the snapshot, a table level lock can often be avoided by utilizing the index. There are scenarios, however, where *any* change to *any* tuple in the relation must be detected. An alternate approach must be adopted in these cases to prevent locking contention. If the underlying data source and application program cannot be modified, an incremental differencing algorithm can be utilized (see section 2.3). If the data source allows for the modification of table schemas, several approaches are available to greatly reduce I/O and improve concurrency.

R* was one of the first database products to augment schemas with a time-stamp that could be used to enhance change detection [Lind86]. R* used the time-stamp for creating snapshots of pre-defined queries. A snapshot in this sense is the same concept as a materialized view in a data warehouse. With the R* approach, each table that is designated as providing data for snapshot is internally augmented with a time-stamp attribute and a record identifier. As tuples are modified, the time-stamp is automatically updated. Although this approach reduces the cost of identifying changes needing application to the snapshot, much of the System R* code was rewritten to properly utilize the new fields. To efficiently detect changes, this approach requires knowledge of every available tuple address per page as the snapshot references the record identifier to determine if deletes have occurred. By their admission, this requirement is quite restrictive. Rewriting the database engine is also not an option for most systems. The R* approach does, however, suggest several techniques that can be adapted to provide good performance for a query-based differencing system.

3.2.1 Simple Time-Stamps (TS)

The basic time-stamp approach has merit if the underlying data source contains table schemas that can be modified to include a time-stamp attribute. The time-stamp field can be either an actual date/time attribute or it can simply contain an integer that is incremented each time a modification is performed. Time-stamps can be used most effectively if the underlying data management system supports auto-incrementing data types. For systems without auto-update capabilities, time-stamps or incremental integers can still be used effectively if all access to the data is restricted to well-defined application interfaces. Controlling database access through the use of pre-defined application interfaces is a common practice in most business environments. In such an environment, application code must be changed so that the time-stamp field is updated at each tuple modification. Inserts can be implemented with null time-stamps or with time-stamps where the appropriate value is pulled from a separate table or a system variable. The non-null time-stamp option leads to a more efficient query by eliminating tri-value logic for nullable key fields.

The problem with this simple approach is its inability to efficiently detect deletions. The application code could insert the tuple into a temporary table before deleting from the base relation, but this unnecessarily lengthens the transaction. Basic time-stamps do not eliminate the need for a copy nor do they prevent the necessity of scanning and joining all rows of the base table and the replicate old copy if deletes are allowed. In this case, the

increased size of each tuple augmented with a time-stamp may hinder instead of improve performance.

3.2.2 Time-Stamps and Delete-Flags (TDI)

To fully utilize time-stamps for efficient change detection, the relation must be additionally augmented with a delete-flag attribute. The accessing application must also be modified to change all *delete* statements to *updates* of the delete-flag field. This marking, similar to the dBase [Dunl90] implementation of mark-then-pack for deletes, enables deleted records to be easily detected. The time-stamp with delete-flag algorithm (TDI) physically removes the marked records, instead of the application, after checking for changes in a separate step. Modifying the application to use a view instead of the base relation easily restricts application reference to non-deleted rows only. The basic algorithm for this approach is similar to that of R* except that it can be applied to any data source with SQL and time-stamp capabilities. Figure 2 outlines this algorithm and Figure 3 shows the performance characteristics of a general-business-use data source using this algorithm.

```
Create cursor CHANGES as
Select t1.f1,t1.f2,…,t1.fn,t1.delete_flag,change_type='Update' from basetable t1,duplicate t2
Where t1.time-stamp>last_time_checked
   And t1.key=t2.key And ((t1.f1!=t2.f1 or t1.f2!=t2.f2 or …t1.fn!=t2.fn) or t1.delete_flag='T')
Union
Select t1.f1,t1.f2,…t1.fn,t1.delete_flag,change_type='Insert' from basetable t1
Where t1.time-stamp>last_time_checked
    And not exists (select t2.key from duplicate t2 where t2.key=t1.key)
Open cursor CHANGES
Fetch CHANGES into structure
While SQLCA.rowcount>0 //stop when no more rows available
  If  structure.delete_flag!='T' {
    // If using with an ATP, send structure to ATP before each "duplicate" modification
    If change_flag='Update' { // Change was an update
      Update duplicate set f1=structure.f1,…,fn=structure.fn where duplicate.key=structure.key }
    Else {  //Must have been an insert
      Insert into duplicate values structure   }
  Else {  //structure is a deleted tuple
       Delete from duplicate where structure.key=duplicate.key
       Delete from basetable where current of cursor  }
Endwhile
```

Figure 2: Time-stamp with Delete-Flag Algorithm to Gather and Forward Changes

The basic TDI algorithm is easy to implement. At defined intervals, the SQL statement in Figure 2 is issued against the data source. The SQL groups changes into two categories, inserts and updates. Two groups are used instead of three to improve the efficiency of the query. An insert is detected if the time-stamp field is greater than the last time the algorithm was run and the same key does not exist in the "replica." An Update is detected if the time-stamp is greater than the last time the algorithm was run and the delete-flag attribute is set to 'T' or the pertinent fields have changed. Recall that the delete flag is set to "T" in the application program in leau of the application code issuing an SQL delete. If the delete flag is set to "T", the update is really a delete and is treated as such. If the "delete flag" is not "T", the change is an update and the values are propagated to the snapshot. The large improvement in performance over previous algorithms is achieved by eliminating the need to

compare rows to identify changes.

Assuming the non-existence of a time-stamp index on B_a, a scan of the base relation is required. Presuming the existence of some type of key on C, the cost of identifying matching records in C is Δ. While the Yao [Yao77] function would argue that changing 1% of the tuples requires accessing more than 1% of the pages of a table, actual implementation suggests just the opposite to be true. If the data source supports large tables, the data source is most likely some form of database product. Products such as Excel or Lotus 1-2-3 are queriable, but the data set size is comparatively small. Even with these products, if the data set spans multiple pages, most changes are grouped due to users adding or removing "sets" of like records at any given interval.

For database products, very few data structures are supported. In the relational arena, the structure of choice is B-tree, with a few vendors also supporting heap, ISAM, and Hash structures [Ingr91][Info97][Micr97][Orac94][Syba96]. With heap structures, all inserts appear at the end of the data set. The records that are most likely to be changed are those that have been recently entered incorrectly or are those that are purged to reduce the size of the table. In either case, the changes are isolated to the last few pages or the first few. ISAM and B-tree structures also place all inserts at the end of the structure if an incremental key is being utilized. While it is possible to have deletes and updates issued against older tuples, most changes to tuples more than one month old are considered changes to historical data. Seldom does historical data need to be changed for business applications. Even if an incremental key is not being utilized, some other attribute can generally be utilized to group changes into categories. As the replicate table can be indexed on any attribute(s) that seem reasonable, the grouping attributes can be used to maintain likely modified tuples on the same page. Keying the replicate table on probable grouping attributes eliminates any distribution problems caused by using a Hash structure on the base relation. For systems, such as personnel applications, where modifications are more random in nature, the number of changes between queries seldom forces access to a large number of pages. Thus, while it is possible for a 1-% change to require access to greater than 1-% of the pages, it is equally likely that such a change will access less than 1-% of the pages.

IO Cost to Identify Changes (scan of B_a) = S
Cost to join B,C to gather old/new pairs - Δ (N)
TOTAL IO Cost TS = S+Δ(N) //assume some type of key on C
PROBLEM: Locking all of B while scan is occurring can degrade concurrency

Figure 3: Performance Analysis of TS&DF Algorithm for a typical business application

To further improve performance of the basic algorithm, the cost of scanning the base relation can be greatly reduced if the base relation is enhanced with a secondary index on the time-stamp attribute (TDI). As an example, Figure 4 illustrates the performance improvement gained by introducing the secondary index. Finally an additional improvement is outlined in Figure 5 where a secondary index is added that contains the time-stamp and the pertinent fields of the snapshot (ATDI).

Placing additional fields in the secondary index increases I/O for updates to pertinent attributes. As "inserts" and "deletes" already affect the secondary, the cost is not appreciably increased for these operations. If all fields in the primary are included in the snapshot, this algorithm performs worse than the simple secondary index approach. For highly restrictive snapshots, however, differencing costs can be greatly reduced. For instance, if the snapshot contains only 30% of the attributes, and a change to 1% of the tuples occurs between each

differencing run, I/O costs can be reduced to .014*N* by eliminating the need to join back to the base relation as shown in Figure 5.

Astute readers will note that the highest cost of the augmented index algorithm is the cost of I/O on the old relation. One way to reduce I/O cost is to restrict the old copy/snapshot to the applicable fields and rows requiring change detection as assumed by the calculation in Figure 5. If both the snapshot and the index are projections of the base relation, the cost of comparison can be greatly reduced if the sum of the size of applicable fields is relatively small. As an example, many relations contain a large text field to hold extraneous data. Eliminating this "comment" field from consideration for update detection when the size of the comment field is greater than the combined size of all other attributes in the relation can reduce the cost of change detection by more than 50%. Figure 6 outlines the cost of utilizing such a technique with the ATDI algorithm for a typical business application.

IO Cost to Identify Changes in Index (Secondary Btree Index on TS) = $I*\Delta$
Cost to join index,B_a = $S*\Delta$ // Need to join index back to augmented base relation
Cost to join *C,B* to gather old/new pairs ≈ $\Delta(N)$ // Need to join identified changes with old Copy
$Cost_{TDI} = S*\Delta + \Delta(N) + I*\Delta + K$
PROBLEM: If a large percentage of the table changes, the cost of using the index can exceed the cost of a base table scan. Note: There is a low likelihood of index-level lock if only 1% changes. It is, however, possible to acquire a table level lock due to the random distribution of changed tuples (Yao function).
Increase in space complexity = *(S+I)-N+N (size of C) = S+I*
For relations where T≤A (highly unlikely,), (S+I)≥3N. Worst case costs with a B-tree is of height K are:
*Max(TDI) = ((I*Δ)+K)+((S*Δ)+K)+((N*Δ)+K)=((I+S+N)*Δ)+3K*

Figure 4: TDI Algorithm Costs using a simple secondary index on TS & DF

I=1+X/(P/(A+size(tid)+K+F)) // Index includes TS, DF, tid, Key, and pertinent fields
Increase in space complexity = (S+I)-N+N = S+I
Cost to identify change in the index = $I*\Delta + K$
Cost to join back to base = 0 – not needed for algorithm
Cost to match with old copy to gather old/new pairs - $\Delta(N)$
$Cost_{ATDI} = I*\Delta + \Delta(N) + K$

Figure 5: ATDI Algorithm Costs using an augmented secondary index

I=1+X/(P/(A+size(tid)+K+F))
Increase in space complexity = *(S+I)-N+W = A+I+W*
Cost to identify change in the index = $I * \Delta$
Cost to join back to base = 0 – not needed for algorithm
Cost to join *I,Cr* to gather old/new pairs - *Δ(W)*
$Cost_{ATDI\_PC} = I*\Delta + \Delta(W) + K$

Figure 6: ATDI Algorithm costs in conjunction with a Projected Copy

3.3 Tuple Level Checksums (TLCS)

Although time-stamps and delete-flags can be utilized to achieve adequate performance, not all applications can be modified to utilize the new attributes. One technique that can be used to improve performance of a system with an unchangeable front-end, like a legacy Cobol application, is that of augmenting each tuple with a checksum field whose value is based upon

pertinent attributes – those attributes having relevance to the snapshot. Use of a checksum value can also be implemented by creating-then-joining to a separate table that contains only the key and the checksum attribute. Modifying the base relation will slightly improve performance by eliminating the need for an additional join.

After the relation's schema is modified, a separate job must be run against a quiet system to calculate and update the value of the checksum for each tuple. As the system becomes active, a differencing procedure is run at specified timer or resource driven intervals to calculate the checksum and compare it to the existing checksum attribute of the tuple. If a discrepancy is detected, the base table checksum is modified and the record is joined to the copy to yield an old/new value pair. The total I/O cost per iteration is

$N+\Delta(W)$ *where* $W < N$

If the number of attributes in C is one-tenth the number in B (a few pertinent attributes), the I/O cost can be quite low. (E.g. it is *1.001(N)* if a 1-% change is assumed and C is appropriately indexed.) If N is less than 100, tuple level checksums perform reasonably well. For tables spanning more than 1000 pages, however, locking the entirety of the base relation prevents the algorithm from being executed on a frequent basis. Although this technique improves upon the previous algorithms where application modification is prohibited, it does not yield sufficient update gathering performance for an asynchronous trigger processor or data warehouse history gathering system due to high locking overhead when large relations are involved. Frequent queries against large tables may maintain an almost constant lock on the relation.

A variant of this approach can be used if the local site does not support joins or if it does not have sufficient disk space for the replicate copy. In such an environment, the change detected by the checksum is queued and the data is forwarded at intervals for joining at the remote site. The queuing reduces message traffic and can improve join performance. Remote site differencing is further explored in section 3.6.

3.4 Incremental Differencing

The Modified Database Schema (MDS) and the Tuple-Level Checksum (TLCS) algorithms both require scans of the base relation to identify changes. This scanning must be performed unless triggers, replication, an intermediate layer modifier, an index, or an application change is introduced. For large tables, scanning the entire relation during peak hours is not a reasonable option due to locking contention. The Incremental Differencing Algorithm (IDA) introduced here presents a plausible solution to this problem.

The basic incremental differencing algorithm is found in Figure 7. The IDA algorithm makes use of an auxiliary table to help in partitioning the data into appropriate subsets. Before the IDA can be started, an initialization step must be run to calculate the number of pages touched per range of clustered key values. This information is stored in the auxiliary relation. The presumption of some form of clustered or ordered key is quite reasonable given that a table spanning thousands of pages does not yield sufficient performance unless some form of key is available. If the base relation is heap with a secondary (non-clustered) index, the number of pages referenced by the p-tids (tid pointers) of the index must be calculated. If the primary or secondary is of hash structure, with no clustering or ordering index type available, the range query parameters generated by the initialization step will be of little use, as a table level (or segment level) scan will be performed for range queries. For this discussion, we assume the existence of a B-tree or ISAM-like indexing structure.

```
Select max_counter=max(counter) from auxiliary a
Where a.table_name = 'my_table';
I=0;
While I<max_counter
  Query data from my_table in the range specified by the start and end
  Values found in row I of the auxiliary relation.  Compare tuples from
  this range to the same range in the older copy of the relation.
  Apply changes to the old relation.
  I=mod(I+1,max_count);
  Wait(x seconds);
}
```

Figure 7: Basic Incremental Differencing Algorithm

The initialization step of the IDA must scan the entire relation looking for values that partition the data into ranges that span a pre-specified number of pages. Partitioning the data by the average cardinality is not sufficient due to the possibility of unequal key distribution. The number of pages in each range (histogram statistics) must be stored to ensure that table-level lock escalation does not occur. As each range is identified, an entry is placed in an auxiliary table with the boundary values of the range as well as an incremental counter to uniquely label each range. After each range tuple is inserted, the page counter is reset, the start-range key value is set to the end-range key value of the previous insert, and the scan of the base relation continues. At the end of the scan, the final range is left with an open-end range to ensure detection of incrementally increasing keys. To simplify the algorithm, the initial start range and the final end range are inserted with the absolute minimum or maximum value available for the data type of the key. This eliminates the need to modify the query within the algorithm for the boundary instances. The auxiliary table is then used in the basic algorithm to formulate successive queries against the base relation. This partitioning of the differencing transaction into subset comparisons in similar to the Mini-Batch technique outlined by Jim Gray [Gray93]. As the distribution of values changes within the relation, the initialization step can be re-run to modify the number of iterations as well as the range values of successive runs. The *wait* function can be modified to be purely timer driven or it can be tuned to incorporate load-balancing information.

3.5 Local Site Incremental Differencing (ID)

While Figure 7 outlines the basic algorithm to incrementally gather subsets of rows of the base relation, it does not detail the comparison to the "old" copy of the relation. The old copy is necessary to determine both the existence and the nature of the change that may have occurred in the specific range. If the local site is such that it allows for the creation of additional relations, both the auxiliary table and the "replicate" copy can be maintained within the same "database" as the base relation. If all tables are maintained locally, a three-way join can be used to determine differences using a modified version of the algorithms in Figure 1 and Figure 7. As each union in the basic Query-Based Differencing Algorithm contains the range specification and a modified "where" clause that references only pertinent attributes, the algorithm does not suffer from locking problems nor does it report spurious modifications. Locking does not escalate due to the key restrictions in the range table. In the worst-case scenario where all attributes must be maintained in the copy, the cost of each complete table comparison is *2N+size (auxiliary table)*. As an example, with a 1 million row table having a

tuple size of *200*, the size in pages of the auxiliary table is (*N/pages per range)/(P/(2*(size (key))+4))*. Assuming a key size of *8* bytes, a range of *10* pages, and a page size of *2048* bytes (i.e. INGRES), the size of the auxiliary is

(100,000/10)/(2000/20)=10000/100=100 pages or approximately .001N

The worst-case cost assuming some type of index on the base relation is *2.001(N)*. This can be reduced at the expense of CPU if the copy is compressed. It is also highly unlikely that the copy need contain all attributes of the original. If using this technique with an ATP and the triggers are simply "on insert" or "on delete", only the key values need be maintained in the copy, thus reducing I/O by *.9N*. Seldom does the list of reasonable triggers within the ATP reference all attributes (necessitating the maintenance of all attributes in the copy). Knowledge of defined triggers can be used to modify the size of the *duplicate* copy.

The overwhelming benefit of incremental differencing is that it can be run even during peak hours to identify changes within a reasonable time window. One significant pitfall is that updates to key values will be incorrectly identified as a delete followed by an insert. This is true, however, for all of the Non-SQL as well as the SQL approaches outlined except the augmented TS/DF algorithm.

The basic incremental algorithm solves many problems and makes differencing on large legacy systems feasible. It does not address situations in which the underlying database schema cannot be modified nor is it appropriate for databases that do not support joins. Some legacy systems do not allow for the introduction of multiple tables into a database. On other systems, even if the modification is technically allowed, the data source does not have sufficient disk space to support both the original and a copy of the base relation. For scenarios such as these, an incremental remote-site differencing scheme is proposed.

3.6 Remote-Site Incremental Differencing with Checksums (ID/RCS)

If the local site does not support more than one table, or if it does not have sufficient space for the duplicate relation, a remote site can be used to store the extra table(s). The remote data site should typically be supported by a join enabled database with good query processing features. At initialization, a partial or full copy of the base relation is shipped to the remote site. The remote copy is then indexed on the keys used in the range queries to efficiently process comparisons when partial snapshots are passed from the base relation. A simple illustration of this approach is found in Figure 8.

While the basic incremental differencing approach of polling subsets based upon values in the auxiliary table is feasible in such an environment, the constant flow of data between sites can degrade system performance by overflowing the network bandwidth. Although the cost of scanning the base relation cannot be reduced (assuming the data source is not modifiable), the number of messages required between the remote and the local site can be lessened if the likelihood of a change within a single range of data is relatively low. Reduced messaging costs between the base and the remote site can be realized by calculating a modified checksum on the pertinent returned fields within the range query and comparing that value with a pre-calculated range-value checksum stored within the auxiliary relation. Jim Gray [Gray93] has suggested the use of block level checksums for data validation in other arenas and Comer [Come91] has outlined a basic checksum algorithm that could be used for such a system. The use of several checksum methods can reduce the likelihood of a missed update.

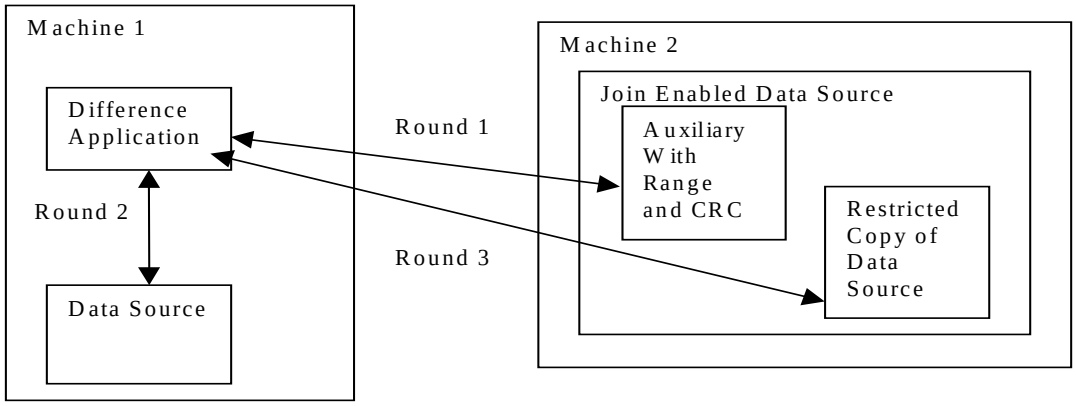

Figure 8: Illustration of Remote-Site Incremental Differencing (RID) Approach

The RID algorithm works as follows. When the range specification is selected from the auxiliary relation to build the query against the base table, the former checksum for that range is also retrieved (round 1). At the source site, the checksum is calculated for data within the range as the data is retrieved (round 2). If the values are identical, no message is returned to the remote site. If a change is detected, the modified checksum value and the tuples within the range are forwarded to the join-enabled remote site to identify the exact nature of the change (round 3). While the Yao [Yao77] function indicates that referencing a randomly selected, small number of tuples quickly escalates to accessing all pages of a table, it does not suggest such an escalation if the selected tuples are identified in a more structured manner. As most applications use some form of incremental key, data is typically structured such that 1-α % of the changes are propagated against the most recent α% of inserts for some α (Zipf's Law). α in this case will typically be 20%. In such a scenario, the use of range checksums will effectively eliminate the need to pass 80% of the relation to the remote site during each incremental relation scan. Moving the tuple comparison and the copy to a remote site also reduces CPU and I/O contention at the source site. The use of multiple sites is especially applicable where data warehousing is implemented.

4.0 View Based Differencing (VBD)

While the query-based algorithms are specifically designed for use with an ATP, their applicability to many other areas should be quite evident. A recent hot topic has been that of data warehousing and view maintenance in data warehouses. Much work has been dedicated to identifying efficient techniques for view maintenance ([Labi97] [Quas96] [Zhug96] [Zhug96a]). In a data warehouse, a materialized view is similar to a snapshot in that it represents a materialized result of a query performed against relations in the source database(s). Refreshing the data in a given warehouse view (snapshot) has been accomplished by using one of two basic approaches. If an application or report is written to calculate a "snapshot," the data is selected, projected, and possibly joined together then output into a file. After file creation, either the entire file is uploaded into the data warehouse or a "diff" is performed against the two versions of the output file to identify changes. Any changes identified are then applied to the warehouse view (snapshot). Without an application, all data

from all involved relations must be copied into flat files. After loading the flat files into the data warehouse, all pertinent snapshots are re-materialized within the remote database. The new view can either replace the older version or it can be compared to the old snapshot to identify changes that must be applied to the old copy. In either case, the cost of managing a single view can be staggering if there are large relations with only a few interesting views.

Query-based differencing can be used to greatly reduce the cost of determining changes by "materializing" the view (query) in a SELECT operation at the base site before forwarding information to the warehouse. Source-site view materialization is especially critical where relationship tables are necessary to complete a many-to-many join between two relations. As the view is processed at the source site, possibly several relations can be eliminated from the copy-then-compare algorithms used in instances where view materialization occurs within the warehouse. Even without time-stamps, the use of range or tuple-level checksums can greatly reduce the I/O required to identify a change that affects a specific snapshot. Because the changes are applied to the warehouse snapshot instead of replacing the entire warehouse view (relation), I/O and downtime within the warehouse are greatly reduced. If the specific warehouse view is concerned with a limited subset of the relations, the reduction in I/O can be quite dramatic. Assuming a 1-% change between each differencing run, the packing of changes into a page before passing to the remote site can reduce messaging costs by greater than 99%.

As an example, consider 4 relations involved in a given view, each containing 100,000 tuples with an average of 10 tuples per page (*N = 10,000*). Without the aide of an application, the cost of identifying and applying the changes to the warehouse, assuming a 1-% change where 50% of the attributes are in the final view, will be quite high. The costs will be either $16N+14N+2(.01(2N)) = 300{,}400$ *I/O* to apply only changes to the view or $16N+4N+2N = 220{,}000$, assuming sort-merge joins are used, to replace the view. These costs are calculated as follows: 4N to read from the source database, 4N to output to a flat file, 4N to read from the flat file, 4N to copy into the remote database tables (the warehouse), 4N to read and join within the database (assuming B-tree structure and ignoring index overhead), 2N to write results to a flat file, 2N to read in the old file, 2N to read in the new file, .01N to write changes to a separate file, .01N to read changes into the database, 2N to read in the old copy from the database, and 2N to write the updated copy back into the database). For view replacement, the second round of writing to a flat file for differencing is eliminated, but the data within the data warehouse is unavailable for a longer period of time.

Several algorithms can be used to facilitate view-based differencing. If local-site view materialization uses a tuple level checksum to identify changes and a keyed structure is available for all tables, the cost of VBD is considerably lower than previous algorithms. Given an efficient structure on the 4 relations being joined, the cost to create a change file is 4.06N (4N to read in the relations plus 4ΔN to join to other relations plus an additional 2ΔN to output the change file). While this cost could be greater given poor storage structure, it could also be less if good secondary indexes are available or some portion of the relation(s) is already cached within the server. The generated change file, of size .02(N) is sent through RPC calls to the warehouse. Applying the changes to the warehouse will require at most 4N (2N for both reading and writing within the warehouse), and it is possible for the costs to be as low as Δ(2N) = .02(N) if the changes are clustered around specific keys. Thus, in the best case, the cost is reduced to $4.1(N) = 41{,}000$, reducing the I/O cost by over *70%*.

If the relations contain time-stamps and delete-flags within a secondary index, a variation of the TDI algorithm (see Figure 2) can be used to efficiently generate a change file that can be applied to the warehouse view. As the changed tuples within each relation must

be joined to the other 3 relations, the total cost of identifying the changes in this example is 4*(3*(.001*1.1)) = .132(N) assuming reasonable clustering of data. As these changes do not *require* that they be written to a flat file, an RPC call to the warehouse can transmit the changes within a few packets. The cost of applying the changes is the same as above with a worst-case cost of 4N and a best-case cost of .02(N). Thus, the "best-case" performance of VBD without restricting the input specifications of the join is .152(N), or 1520 I/O. If the snapshot is a highly restrictive one, the performance improvement is even greater. When dealing with gigabytes of data, this improvement can reduce the time required to reload the warehouse from hours to only a few minutes.

5.0 Performance Evaluation

In the previous sections, several new algorithms were introduced to support frequent snapshot differencing: a modified database-schema approach (MDS), three different modified table-schema approaches (ATDI, TDI, TS/TLCS), and two interval differencing techniques (ID/RCS, ID). A query-based algorithm was also introduced to support view-based differencing (VBD) for systems such as data warehouses. The cost of each approach was compared to previous algorithms having best case performance of 2N I/O, and more likely yielding a cost of 6N or even 8N I/O [Labi96]. The results can be found in figures 9 and 10. To simplify the graphs, only a representative algorithm from each category was chosen. A category is determined by the method used to identify the changes (i.e. comparing flat files, joining tables, using secondary indexes, using time-stamps, etc.)

Concurrency and I/O are only two of the issues that need to be considered when selecting an appropriate change detection algorithm. Figure 11 lists many of the issues that must be considered and ranks the data gathering techniques on a scale from 1 to 5. While the scale is fairly arbitrary, the comparative differences are important. No single solution is best for all data sources, especially considering the limited availability of some of the least costly alternatives. If the difference gathering system is designed in an extensible way, all of these techniques can be made available to the "differencing DBA" – the database administrator in charge of change detection and data replication. If the differencing is linked to an Asynchronous Trigger Processor (ATP), the trigger definitions can be utilized with several of the techniques to further improve differencing performance.

Figures 9 and 10 show that the cost of data gathering varies widely between different techniques. The large fluctuation is essentially due to the discrepancy in the capabilities of the underlying data source. For the same type of data source, however, the new algorithms consistently outperform all previously published differencing techniques of which we are aware. The amount of improvement realized is dependent upon the data source and the ability to change the program code of the accessing application.

The best previously published algorithms for gathering differences in structured data are the pre-sorted and sliding window protocols that compare two flat files [Labi96]. For our example of a 1000 page relation, the cost of comparison was given as 2000 I/O. There are two problems with this estimate. The first issue is that this estimate does not include the cost of generating the flat file. A report must be run to create the separate file from the underlying data source on all but a very few sources of data. The cost to generate the file will typically be at least 2N as the data must be read and then written out. As a presorted order is necessary for the pre-sort algorithm, sorting the relation may cost an additional 4*N*. The sliding window protocol is reasonable if a report is run to generate the file. However, simply dumping the data from a relation cannot be guaranteed to yield consistent ordering, so the sorting cost

cannot be ignored. Thus, a more reasonable estimate for the change detection costs of the previous algorithms is at least 6*N*. Just as important, the algorithms of [Labi96] and other previous techniques cannot be run against active systems due to the number of locks acquired during the report generation or comparison phase. In either case, the graphs in Figures 9 and 10 show that the I/O costs are high for all percentages of tuples and attributes that may be of interest.

The slowest new algorithm proposed, MDS, yields a worst-case cost of 4N, and more likely yields a cost of 2N. With this approach, the data source contains both the copy and the original and performs a sort-merge outer-join to identify changes. This technique is applicable for data sources that do not allow table schema modification but do allow for the introduction of new relations. It is interesting to note that restricting the percentage of attributes in the duplicate relation can improve performance, but restricting the percentage of tuples found in the duplicate can have a dramatic affect on I/O due to the benefit of reducing I/O in both the duplicate and the base relation.

Most of the other new techniques that do not allow for application modification yield performance results with I/O costs approximating N for the average case. The cost curve of ID/RCS demonstrates this assuming a 1-% change per iteration. The improvement in costs is achieved by comparing only interesting tuples instead of reviewing all tuples for the existence of a change.

If the underlying application can be modified, a dramatic reduction in I/O can be realized by eliminating the need to compare tuples to determine the existence of a change. The I/O cost, as is seen by the ATDI curve in Figures 9 and 10, can be reduced to as low as .01N. Applications could be modified to insert a change record into a separate table or tables, but this necessarily doubles the length of every transaction. The number of I/Os, locks, and messages all double with this format (with a few possible exceptions like Oracle where tuples from multiple relations may be found on one page). With the ATDI approach, I/O within the application is not appreciably affected.

If a program is written to detect changes and the change detection is restricted to a subset of the tuples or attributes, the program(s) requires modification whenever a different subset is required. In addition, unless the data source supports triggers, the application must select each row of data that is to be modified if an old/new value pair is to be output. With the use of time-stamps and delete flags, the program(s) need only be changed once as the SQL that performs the differencing is dynamically created based upon trigger specifications. Once the program has been changed to update the time-stamp and delete-flag fields, no other code modification is required. The size of each transaction remains essentially unaffected. As the differencing algorithm makes use of information regarding snapshots to restrict the *where* clause, any number of *views* can be altered and maintained by simply creating or modifying an externally defined trigger. Successive iterations of the differencing algorithm yield old/new value pairs as output. If the time-stamp and delete-flag can be placed in a secondary index, the cost of I/O for each differencing scan can be greatly reduced by eliminating the need to compare attributes to determine the existence of a change. Although this technique is not a pure "differencing" algorithm, it is included here to show the dramatic improvement in performance that can be obtained by using the capabilities of the underlying data source.

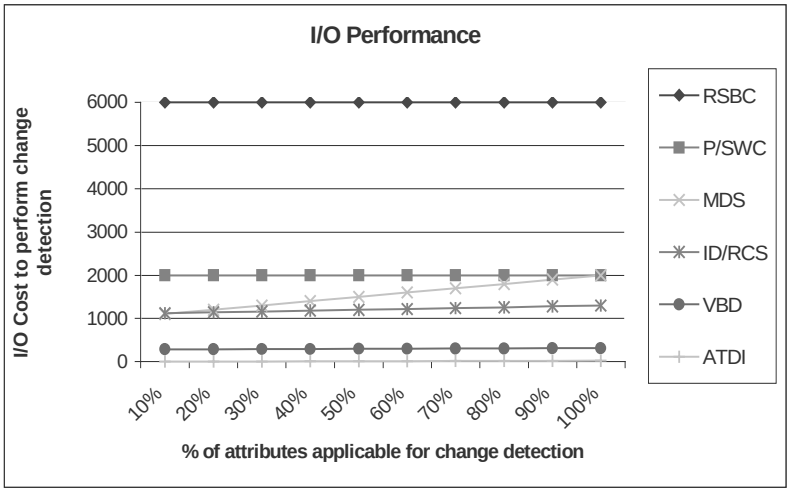

Figure 9: Performance Evaluation Varying the Percentage of Attributes that are required in the older copy for comparison

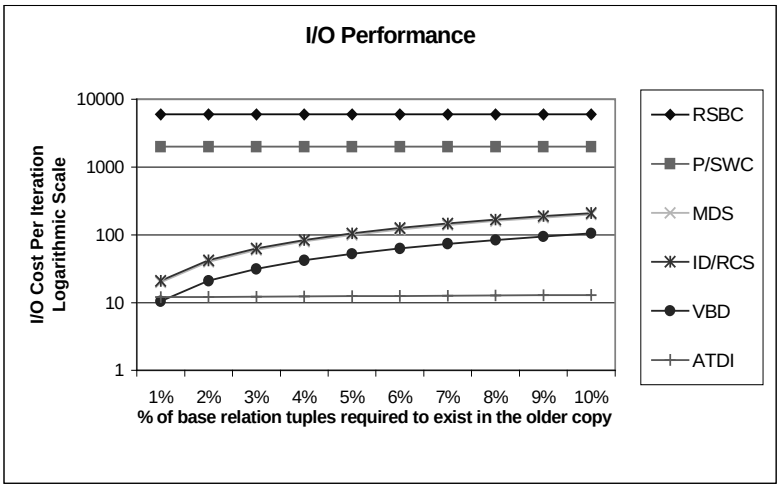

Figure 10: Performance Evaluation Varying the Percentage of Tuples that are required in the older copy for comparison.

Label	Algorithm Description	Cost Formula
RSBC	Report before comparison with sliding window – average previous algorithm	6N
P/SWC	Best case for previous algorithms – presorted/sliding window protocols	2N
MDS	Modified Data Source algorithms performing joins	R(N)+M
ID/RCS	Incremental Differencing with Range Checksum	R(N)+size(Auxiliary)+Δ(M)
VBD	View Based Differencing	Z*((N/Z*(size(key)+A))+Δ(N))+Δ(M)
ATDI	Augmented time-stamp with delete-flag Index	X/(P/(A+size(tid)+F)))*Δ+Δ(M)

Table 3: Legend Definitions for Figures 9 & 10

	IO COST	OLTP SUPPORT	WIDELY AVAILABLE	SPACE COMPLEXITY	MESSAGE COST	TRANSACTION SUPPORT	NOTIFICATION TIME	FREQUENT DIFF SUPPORT	CATCH EVERY UPDATE	SELECTED CHANGE ONLY?	GOOD FOR LARGE TABLES	GOOD FOR 1%CHANGE	GOOD FOR 30% CHANGE	SUPPORT FOR VIEW DIFF	USES PRE-CACHED DATA
REPLICATION	2	1	5	2	1	2	2	0	2	1	1	1	1	5	0
TRIGGERS	1	2	4	1	1	1	1	0	1	1	1	1	4	4	1
ODBC UPD TRAP	1	1	5	1	2	1	1	0	3	2	1	1	1	4	0
FLAT FILE DIFF	5	4	1	4	4	4	5	4	5	5	4	4	4	5	5
TS	3	5	2	3	1	3	4	4	4	1	4	3	3	3	3
TDI	2	3	3	3	1	3	3	2	3	1	1	1	3	1	2
ID	3	2	2	3	2	3	4	2	4	1	3	3	3	2	3
ID/RCS	3	2	2	3	3	3	4	2	4	2	2	2	4	4	3

Ranking from 1 to 5 with 1 being the best and 5 being worst. 0 indicates not applicable.

Figure 11: Qualitative Comparison of Update Gathering Techniques with Respect to Relevant Issues

If the number of I/O cannot be substantially reduced, the impact of table-level locking for larger tables prohibits running the differencing algorithms during highly active periods. While yielding good I/O performance, the two incremental differencing approaches are introduced as a means of reducing locking contention when referencing large data sets. Through the use of time-stamps or range level checksums, the cost of change detection remains below 2N.

The reduction in I/O is even more evident when comparing View Based Differencing (VBD) to previous techniques. By joining the relations within the data source, the need to replicate all relations involved in the view is eliminated. The joins act as natural filters of other relations to reduce the number of tuples that need to be copied. As a snapshot does not generally contain all of the attributes of all involved relations, a further reduction in I/O, as compared to copying all attributes of all relations to the warehouse, is accomplished by the projections on the snapshot. Finally, as some relations involved in the view may serve only to complete the join, joining at the base site eliminates the "intermediate" relation entirely from the copy process.

6. Conclusions and Future Work

Although the performance gains of the new algorithms, especially ATDI, are quite dramatic, perhaps the most important results yielded by these algorithms are those of eliminating locking contention and reducing the need for writing many differencing applications. Previous algorithms could generally be run no more frequently than once a day due to the extreme amount of I/O and the locking contention placed on the applicable relations. For most systems, the algorithms also required an independent application to be written before any differencing could be performed. The new algorithms introduced here solve these issues in one of two ways. If the snapshot is highly restrictive, pushing the restrictions into the SQL statement eliminates the propagation of spurious information.

Restricting the size of the "replica" to pertinent tuples and attributes further reduces I/O costs. If the snapshot is a fairly complete replica of the original, locking contention is avoided by partitioning the differencing into smaller ranges and performing incremental queries. In either case, the actual implementation can be accomplished by simply defining an SQL statement instead of writing an entire application if the underlying extensible differencing application is already completed.

As these algorithms are designed for use with an asynchronous trigger processor or a data warehousing system, there are several issues that must still be addressed. As future work, we are considering restricting the snapshot and the query of the original site to be based upon information contained within the trigger specification(s) or the view specification(s). While the algorithms perform well for smaller sets of triggers against a given relation, the introduction of a large number of views or triggers can lead to a large number of overlapping queries. Repetitive queries with simple variable replacement in the "where" clause can seriously degrade system performance. Synthesis of trigger definitions or view specifications for input into the differencing system can lead to improved performance by reducing the number of queries against the base relations. Quass [Quas96] has done some interesting work on optimizing data warehouse self-maintainability, but that work does not address the synthesis of trigger conditions. If the conditions can be synthesized and placed into a relation, the differencing algorithm can make use of the information by dynamically modifying the differencing query.

REFERENCES

[ACT96] A Joint Report by the ACT-NET Consortium, "The active database management system manifesto: A Rulebase of DBMS features," *Communications of the ACM*, Vol. 25, No. 3, September 1996, Pages 40-49.

[Alon90] Alonso, Rafael, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," *Proceedings of the ACM* Transactions on Database Systems, Vol. 15, No. 3, September 1990, Pages 359-384.

[Bern90] Bernstein, P., "TP Monitors," *Communications of the ACM*, Volume 33, No. 11, November 1990, Pages 75-86.

[Bern90b] Bernstein, P., M. Hsu, B. Mann, "Implementing Recoverable Requests Using Queues", *Communications of the ACM,* Volume 19, No. 2, November 1990, Pages 112-122.

[Brow97] Brown, L., *Oracle Database Administration*, Prentice Hall Publishers, 1997.

[Chaw96] Chawathe, S., A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," *Proceedings of the ACM* International Conference on the Management of Data, Volume 25, No. 2, June 1996, Pages 493-50.

[Chaw97a] Chawathe, S., S. Abiteboul, J. Widom, "Representing and Querying Changes in Semistructured Data," Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida, February 1998, Pages 4-13.

[Chaw97b] Chawathe, S., H. Garcia-Molina, "Meaningful Change Detection in Structured Data," *Proceedings of the ACM* International Conference on the Management of Data, Volume 26, No. 2, June 1997, Pages 26-37.

[Come91] Comer, Douglas E., *Internetworking with TCP/IP,* 2nd Edition, Prentice-Hall, Inc., 1991.

[Date93] Date,C.J., and Hugh Darwen, *A Guide to the SQL Standard,* 3rd Edition, Addison Wesley, 1993.

[Dunl90] Dunlop, N., *DBASE for professionals,* New York: Van Nostrand Reinhold, 1990.

[Geig95] Geiger, K., *Inside ODBC*, Microsoft Press, 1995.

[Gray93] Gray, Jim, *Transaction Processing, Concepts and Techniques*, Morgan Kaufmann, 1993.

[Hans97] Hanson, Eric, and Khosla, Samir, "An Introduction to the TriggerMan Asynchronous Trigger Processor," Proceedings of the 3rd International Workshop on Rules in Database Systems, Skovde, Sweden, June 1997, Pages 51-66.

[Info97] "Informix Universal Server," http://www.informix.com

[Info97a] Informix, "Developing DataBlade Modules for INFORMIX-Universal Server, http://www.informix.com

[Ingr91] Ingres, "Ingres SQL Reference Manual," Ingres Corporation, 1991.

[Labi96] Labio, W., H. Garcia-Molina, "Efficient Snapshot Differential Algorithms for Data Warehousing," Proceedings of VLDB, September 1996, pp. 63-74.

[Labi97] Labio, W., D. Quass, B. Adelberg, "Physical Database Design for Data Warehouses," Proceedings of the 13th International Conference on Data Engineering, April 1997, Pages 277-288.

[Lind86] Lindsay, B., L. Haas, C. Mohan, H. Pirahesh, P. Wilms, "A Snapshot Differential Refresh Algorithm," *Communications of the ACM*, Volume 15, No. 2, 1986, Pages 53-60.

[Micr96] Microsoft, *Microsoft SQL Server*, Microsoft, 1996.

[ODBC94] *ODBC 2.0 Programmer's Reference and SDK Guide*, Microsoft Press, 1994.

[Orac94] Oracle, "Oracle 7 Symmetric Replication," Oracle White Paper, June 1994.

[Penn97] Penner, R., "ODBC 3.0: What's New?," Unix Review, April 1997.

[Quas96] Quass, D, A. Gupta, I. Mumick, J. Widom, "Making Views Self-Maintainable for Data Warehousing", Proceedings of the 14th Conference on Parallel and Distributed Information Systems, December 1996, Pages 158-169.

[Syba94] Moissis, A. "Sybase Replication Server," Sybase Technical Paper Series, 1994.

[Syba96] Sybase Replication Server Technical Overview, Sybase Inc., 1996.

[Ull89] Ullman, J.D., *Principles of Database and Knowledge-Base systems*, Computer Science Press, Rockville, MD, 1989.

[Usoft96] Usoft, "Business Rules Automation", Usoft White Paper, 1996, http://www.usoft.com/whitepapers.

[Yao77] Yao, S., "Approximating Block Accesses in Database Organizations," *Communications of the ACM*, Volume 20, Number 4, April 1977, Page 260.

[Zhug96] Zhuge, Y., H. Garcia-Molina, and J. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency," Proceedings of the 14th Conference on Parallel Distributed Information Systems," December 1996, Pages 146-157.

[Zhug96a] Zhuge, Y., H. Garcia-Molina, and J. Wiener, "Consistency Algorithms for Multi-Source Warehouse View Maintenance," Journal of Distributed and Parallel Databases, Volume 6, No. 1, July 1997, Pages 7-40.